Techniques for Object Oriented Analysis and Design

I've been working in Object Technology for over a decade now, and have learned many things about the analysis and design of object-oriented information systems. It's an enjoyable field to work in, and one that still passes on new lessons every year. These pages distill some of the lessons that I've learned over this time.

These days such techniques are dominated by the Unified Modeling Language (UML), which is now the standard modeling language for object-oriented development. If you use any of the older techniques, I strongly recommend that you shift to the UML now, as it is clearly going to be the dominant notation system. If you are looking to start working with a design notation, the UML is the one start with, as it already is dominating the industry.

There is one caveat to using the UML, however. There are still a lack of really good tutorial books on the UML. Of course I have to mention my own book, UML Distilled, but that is a brief guide for someone who already knows about OO analysis and design and wants a brief overview of the UML. It is not a tutorial book. What I have seen of Grady Booch's forthcoming book looks good, but it is not due to be published until the end of 1998.

These pages are separated into three broad sections. The first bit looks at several general topics for analysis and design techniques: the elements of a technique, standardization and the UML, case tools, and why you might bother with a technique at all. I follow this with several pages on different modeling techniques such as class diagrams, interaction diagrams, and use cases. The final section looks at process techniques, such as evolutionary delivery, patterns, and refactoring. The techniques I talk about are both UML and non-UML techniques. Although the UML is a the dominant modeling language, there are important techniques that exist in addition to the UML, and I don't hesitate to use them when I need to.

Please let me know how what you feel about this site, I hope it can evolve into a leading source of information on OO methods. In addition if you have any questions about the techniques, don't hesitate to get in touch. I'm always happy to answer emails. I do training and consulting on these techniques, let me know if you are interested. My book UML Distilled goes into more detail on many of these issues, so if you find these pages handy, you might take a look in there.

Contents

- Modeling Techniques
  - Use cases
  - Class Diagrams
  - CRC Cards
  - Package Diagrams
  - Design by Contract
  - State Diagrams
  - Interaction Diagrams
  - Activity Diagrams
- Process Techniques
  - Incremental Development
  - Translation
  - Patterns
  - Refactoring

|home|

Standardization and the UML

A few years ago we saw many different methods being pushed by their respective gurus, and users were in a tizzy as to which one to choose. Standardization seemed impossible, most gurus claimed it was undesirable. Now we hear about the UML as the new standard in methods. Is this the whole story and how does this affect you?

UML and standard notation but not a *method*

The Unified Modeling Language (UML) is a standard notation for modeling object-oriented systems. It has the formal support of the Object Management Group (OMG) and its various member companies, which pretty well means everyone in the OO software business.

It's important to realize, however, that the UML is only a standard *notation*. Essentially it defines a number of diagrams that you can draw to describe a system, and what these diagrams mean. It does not describe the process you use to go about building software. Such a process description, or *method*, would include a list of tasks that need to be done, what order they should be done in, the deliverables produced, the kinds of skills required for each task etc. The traditional formal methodology consists of both notations and a method.

The idea is that by standardizing on the notation, software developers can better communicate providing all the deliverables in a method will use the UML. However different groups are free to use whichever method they want to use to actually go about building software. Several methods have been proposed that use the UML. Rational have published their Objectory method (which I've heard is now to be called the Unified Process), strongly based on the work of Ivar Jacobson. HP's Fusion method is another method that is widely talked about.

Should you use the UML?

In a word - yes. The older OO notations are dying out at a rapid pace. They will still linger in books published before the UML stabilized, but newer books articles and the like will all use the UML as notation. If you are using an older notation, you should convert to the UML during 1998. If you are just starting to use modeling notation, you should learn the UML directly.

However you need to be aware that the UML is awfully complicated. This is because it tries to provide diagrams for every sort of situation you need to deal with. So if you are starting with the UML you need to be selective. Concentrate on using just a little of the notation. A good start is to use class diagrams and interaction diagrams. Even with class diagrams start with the basic notation, don't try to go for all of it. Also don't limit yourself to only the UML for useful techniques. Such techniques as CRC cards, while not part of the official UML, are still well worthwhile.

As for a method, there it depends on what you are doing and how much ceremony you like in your

software process. Look for a method that is designed with your kind of software in mind. A process designed for real-time systems is not likely to work well for business information systems. There is a strong current in favor of a more minimal methodology, trying to cut down the ceremony of the traditional methods. This is particularly appropriate for small and medium scale developments with a couple of dozen developers or less.

You can get free information on the UML from the official UML web site, but its turgid material meant for method specialists rather than the regular developer. Books are beginning to appear. I haven't made a full survey of the new UML books so I can't comment on them here. Even if I did, it would hardly be an unbiased survey as I came out with the first UML book (UML Distilled). The older methods books are slowly getting changed to UML notation, for example Jim Odell's foundations book.

The history of how we got here

There were several attempts a few years ago to do some standardization efforts in OO methods. An Analysis and Design Special Interest Group (SIG) of the Object Management Group (OMG) met for a couple of years, published a thick report of OO methods, and was widely ignored. Except that is for an open letter criticizing the very idea of methods standardization, signed by the major gurus. Grady Booch tried a more informal tack, but over breakfast at OOPSLA 93 he found that the other gurus were not really all that interested. I remember a panel at OOPSLA 93 where the gurus said that while it might be a good thing, nobody was likely to put any effort into it.

The event that shook this little world was Jim Rumbaugh, the lead guru of OMT, joining Rational (where Grady Booch is Chief Scientist). They promised to work together to create a Unified Method. As two of the leading methodologists, a combined Rational Unified Method would have a lot of weight.

That event sent methodologists into a flurry of panic. At one point Rational were saying they were going to achieve standardization the Microsoft way. Grady and Jim announced that the methods war was over: "we won". Elsewhere methodologists formed an Anti-Booch Coalition. It was all very amusing.

More importantly in June 1995 the OMG restarted the Analysis and Design SIG. In its earlier incarnation it worked in its own little corner and had got little attention from the OMG management. Now the OMG was taking it seriously and Mary Loomis and Jim Odell were made co-chairs. Clearly the OMG did not want Rational setting the standards without some outside influence. The SIG prepared an RFP (request for proposal) for standards: interfaces and semantics for modeling together with a notation.

By OOPSLA 1995 Rational were ready to issue their first document on the Unified Method (version 0.8) at an enormously popular fringe event which, despite Jim Rumbaugh's singing, was actually good fun. But Rational also had another shock up its sleeve: the purchase of Ivar Jacobson's company Objectory. With three top methodologists under Rational's wing, how could they fail to impose a standard?

During 1996 Rational blended Ivar Jacobson's ideas into the Unified Method, changing its name into the *Unified Modeling Language* (UML). They also began to become more open to other views, and received a lot of endorsements from big name companies like Hewlett-Packard, Microsoft, Oracle, and Texas Instruments.

In early 1997 several submissions were made to the OMG Analysis and Design Task Force. Rational's

was the one that got the most attention, but there were also several other submissions from other organizations, most notably that from IBM/ObjectTime. At this point the whole process went into the smoke-filled room stage, and there was much manipulating of the meta-model, horse trading, and other forms of sausage-meat. The important result was that all the submitters ended up converging on a single UML.

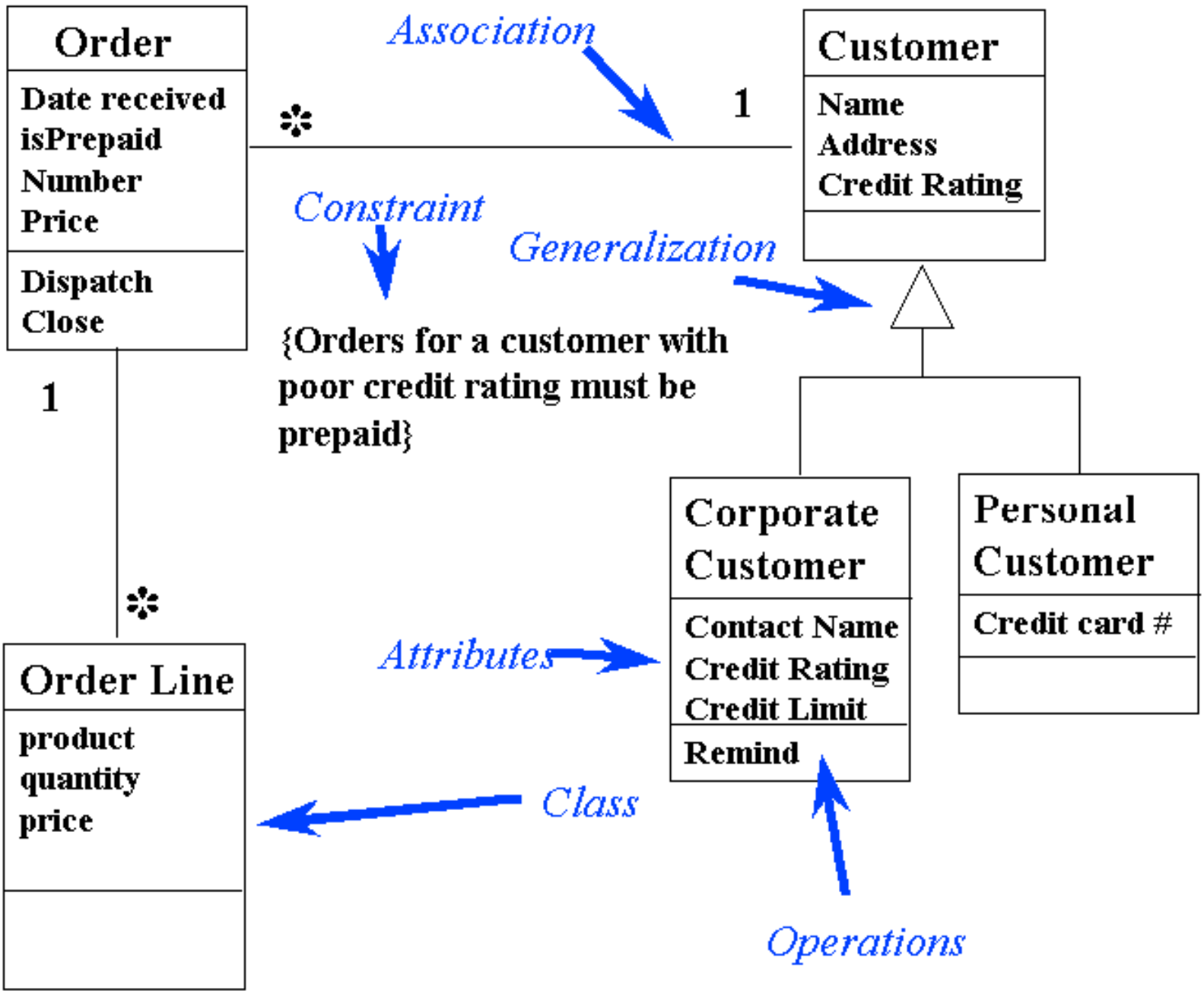
Not all methodologists agreed with headlong rush to the UML. The most open dissenters have been the OPEN consortium, led by Don Firesmith, Brian Henderson-Sellers, and Ian Graham. They claim that the UML is too rooted in old-fashioned data modeling techniques, which are unsuitable for object modeling. They came up with a rival: the Open Modeling Language (OML).

But the UML dominated the picture. By mid 1997 all the OMG submitters had converged on UML version 1.1. This was formally adopted by the OMG at the end of 1997. In one last burst of confusion the OMG took the Rational version 1.1 and called it OMG version 1.0. Why they did such a confusing thing, I have no idea. Currently a working group is cleaning up the spec and coming up with version 1.2 (in both numbering schemes). There are no significant changes from version 1.1 to 1.2.

|contents|next|

Class Diagrams

The class diagram is a central modeling technique that runs through nearly all object-oriented methods. This diagram describes the types of objects in the system and various kinds of static relationships which exist between them. There are three principal kinds of relationships which are important: associations (a customer may rent a number of videos), subtypes (a nurse is a kind of person) and aggregation (an engine is part of an aircraft). The various OO methods all use different (and often conflicting) terminology for these concepts, this is extremely frustrating but inevitable: OO languages are just as inconsiderate. It is in this area that the UML will bring some of its greatest benefits in simplifying these different diagrams. In this section I will use the UML terms as my main terminology, and relate to other terms as I go along.



Perspectives

Before I begin describing class diagrams I have to bring out an important subtlety in the way that people use class diagrams. This is a subtlety that is usually undocumented, but has an important impact on the way you should interpret a diagram, for it really concerns what it is you are describing with a model. Following the lead of [Cook and Daniels] I say that there are three perspectives you can use in drawing class diagrams (or indeed any model, but it is most noticeable in class diagrams).

- *Conceptual:* In this case you are drawing a diagram that represents the concepts in the domain under study. These concepts will naturally relate to the classes that implement them, but it is often not a direct mapping. Indeed the model is drawn with little or no regard for the software that might implement it, and is generally language independent. ([Cook and Daniels] call this the essential perspective, I use conceptual as the usage has been around for a long time)
- *Specification:* Now we are looking at software, but we are looking at the interfaces of the software, not the implementation. We are thus looking at types rather than classes. Object-oriented development puts a great emphasis on the difference between type and class, but this is often overlooked in practice. In my view it is important to separate interface (type) and implementation (class). Most OO languages do not do it and methods, influenced by that, have followed suit. This is changing (Java and CORBA will have some influence here) but not quickly enough. Types represent an interface which may have many implementations due to implementation environment, performance characteristics, or vendor. The distinction can be very important in a number of design techniques based on delegation, hence the discussion in [Gang of Four]
- *Implementation:* In this view we really do have classes and we are laying the implementation bare. This is probably the most often used perspective, but in many ways the specification perspective is often a better one to take.

Understanding the perspective is crucial to both drawing and reading class diagrams. As I talk about the technique further I will stress how each element of this technique depends heavily on the perspective. When you are drawing a diagram, draw it from a single clear perspective, when you read a diagram make sure you know which perspective the drawer drew it in. That knowledge is essential if you are to interpret the diagram properly. Unfortunately the lines between the perspectives are not sharp, and most modelers do not take care to get their perspective sorted out when they are drawing.

Associations, attributes and aggregation

Associations represent relationships between instances of types (a person works for a company, a company has a number of offices…). The interpretation of them varies with the perspective. Conceptually they represent conceptual relationships between the types involved. In specification these are responsibilities for knowing, and will be made explicit by access and update operations. This may mean that a pointer exists between order and customer, but that is hidden by encapsulation. A more implementation interpretation implies the presence of a pointer. Thus it is essential to know what perspective is used to build a model in order to interpret it correctly.

Associations may be bi-directional (can be navigated in either direction) or uni-directional (can be navigated in one direction only). Conceptually all associations can be thought of as bi-directional, but uni-directional associations are important for specification and implementation models. For specification models bi-directional associations give more flexibility in navigation but incur greater coupling. In implementation models a bi-directional association implies coupled sets of pointers, which many designers find difficult to deal with.

One of the key aspects of associations is the cardinality of an association (sometimes called multiplicity). This specifies how many companies a person may work for, how many children a mother can have, etc. This corresponds to the notion of mandatory, optional, 1-many, many-many relationships in the Entity-Relationship approach. The

UML has standard the mass of different approaches that existed here.

Generalization

Subtyping is the most obvious addition to ER diagrams for use in OO. It has an immediate correspondence to inheritance in OO programming. However the object-oriented community should not forget that subtyping has been around in data modeling long before the object cavalry rode over the horizon.

A typical example of subtyping is to consider personal and corporate customers of a business. They have differences but also many similarities. The similarities can be placed in a general customer class with personal and corporate customer as subtypes.

Again this phenomenon has different interpretations at the different levels of modeling. Conceptually we can say that corporate customer is a subtype of customer if all instances of corporate customer are also, by definition, instances of customer. From a specification model we would say that the interface of corporate customer must conform to the interface of customer. That is an instance of corporate customer may be used in any situation where a customer is used, and the caller need not be aware that a subtype is actually present (the principle of substitutability). The corporate customer may respond to certain commands differently than another customer (polymorphism) but the caller should not need to worry about the difference.

Inheritance and subclassing in OO languages is an implementation approach. It says that the subclass inherits the data and operations of the superclass. It has a lot in common with subtyping, but there are important differences. Subclassing is only one way of implementing subtyping (see [Odell pragmatics] or [Fowler]). Subclassing may also be used without subtyping and most authors frown upon this practice. Newer languages and standards increasingly try to emphasize the difference between interface-inheritance (subtyping) and implementation-inheritance (subclassing).

Constraint Rules

One area of OO analysis and design that has gained more attention in recent years is that of rules. The general notion is to apply the ideas of AI on rule based systems into OO modeling. Actually some of these have been around for a while in the object community. Eiffel has long had support for assertions as part of its principle of Design by Contract, which been sadly neglected in many OO methods. In the structural view the principal assertion is the constraint. This is a logical expression about a type which must always be true. Cardinality express some constraints, but not all. UML uses the brace {} notation to show constraints on the structural model.

When to Use Them

Class diagrams are the backbone of nearly all OO methods so you will find yourself using them all the time. The trouble is that they are so rich that they can be overwhelming to use. Here are a few tips:

- Don't try to use all the various notations on offer. Start with the simple stuff: classes, associations, attributes, and generalization. Introduce other notations only when you need them.
- Sort out which perspective you are drawing the models from. If you are in analysis draw conceptual models. When working with software concentrate on specification models. Draw implementation models only when you are illustrating a particular implementation technique
- Don't draw models for everything, concentrate on the key areas. It is better to have a few diagrams that you use and keep up to date than many forgotten, out-of-date models.

The biggest danger with class diagrams is that you can get bogged down in implementation details far too early. To

combat this use the conceptual or specification perspective. If you get these problems you may well find CRC cards to be extremely useful.

Where to Find Out More

At the moment my advice depends on whether you prefer an implementation or a conceptual perspective. For an implementation perspective try [Booch], for a conceptual perspective try [Odell foundations]. Once you have read your choice read the other one. Both perspectives are important. After that any OO book will add some interesting insights. I particularly like [Cook and Daniels] for its treatment of perspectives and the formality that they introduce.

|previous|contents|next|

Annotated Bibliography

When I started doing this kind of comparison I would read every book on OO analysis and design, and most of the papers on the subject. I had to abandon that a long while ago - there is just too much material. This is my personal annotated bibliography of works I have read and found particularly interesting and useful.

[Beck]

Beck, K. Smalltalk Best Practice Patterns. Prentice Hall, Englewood Cliffs, NJ, 1997.

Not strictly a book on analysis and design techniques but useful as many of its patterns delve into the sparsely documented field of refactoring. Many of the patterns are useful for other OO languages too, so its worth reading even if you aren't a smalltalker.

[Beck and Cunningham]

Beck K and Cunningham W "A laboratory for teaching object-oriented thinking" Procedings of OOPSLA 89

The paper that introduced CRC cards.

[Booch]

Booch, G. Object-Oriented Analysis and Design with Applications,(Second Edition Addison-Wesley, Menlo Park CA, 1993.

If you are looking for a book to introduce you to object-oriented design, few would disagree that this is a worthy place. The books has several chapters describing OO concepts, a description of his notation and methodology, and then several examples which discuss how to use the techniques in C++. As one of the authors of the UML his ideas have taken a leading place in the development of the UML. The biggest disadvantage of the book is that it has very much an implementation perspective, so it is good to use [Odell foundations] as a counter-weight. It is also not in UML notation, but the ideas are still worthwhile.

[Booch solutions]

Booch, G. Object Solutions: Managing the Object-Oriented Project, Addison-Wesley, Menlo Park CA, 1996

This is Booch on project management, and a good read it is too. He emphasizes the evolutionary process with an emphasis on actively attacking risk. He adds many war stories and a bunch of reasonable rules of thumb. It is also not too long. It is not as comprehensive as [Goldberg and Rubin] but it is a fine read nonetheless.

[Buschmann et al]

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M, Pattern-Oriented Software Architecture: A System of Patterns, Wiley, Chichester UK, 1996

Patterns collected by several developers at Siemens. Includes several architectural patterns as well as design patterns.

[Coad, North, Mayfield]

Coad, P., North, D. and Mayfield, M. Object models: strategies, patterns, & applications, Prentice Hall, 1995.

Coad's take on patterns. These are different to the PLoP style. They are typical combinations of models. The key to the book is chapter 6 where Coad explains how to use his patterns to help generate a model. This approach begins by identifying certain kinds of easy-to-spot objects. For I.S. systems these are containers and transactions, for real-time systems he starts with devices, and then containers and transactions. The patterns describe typical configurations of classes, so you can look at all the patterns for containers and see where they might apply in the problem. The patterns then spread out from the various players. Each pattern also suggests typical behaviors.

http://www.oi.com/oi_home.html

[Cook and Daniels]

Cook, S. and Daniels, J. Designing Object Systems: object-oriented modeling with Syntropy, Prentice Hall International, Hemel Hempstead, UK, 1994.

One of my favorites. Cook and Daniels wanted a rigorous method with lots of formalism, yet to hide the formality behind OMT style diagrams. For those who want formality this is the best book. Even if you don't want formality their ideas and rigor will be a valuable read. It is particularly valuable for their discussion of perspective in class diagrams and is the best source information on Harel statecharts. It is an advanced book, however, so I would not suggest it as a first read.

[Cockburn]

Cockburn, A. Surviving Object-Oriented Projects, Addison Wesley, 1997

A great book that concentrates on project management, with a great emphasis on incremental development. Everyone leading an object project should read this book.

[Fowler]

Fowler M, Analysis Patterns: Reusable Object Models, Addison-Wesley, Reading MA, 1997

Extracts of domain models that I have come across in developing object-oriented software with examples from healthcare, financial trading, accounting, planning, and layered architectures.

[Fowler, UML]

Fowler M with Scott K, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, Reading MA, 1997

A concise (under half and inch) guide to the new UML. Written for those who already understand the basics of OO modeling and want a quick overview of the important bits of the UML.

[Gang of Four]

Gamma E, Helm R, Johnson R, and Vlissides J, Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, Reading MA, 1995

The first flowering of the patterns movement and one of the most important software books published in the last ten years. A catalog of 23 common designs with rationale, models, and example code. If you are at all serious about OO programming you must have this book.

[Goldberg and Rubin]

Goldberg A. and Rubin K., Suceeding with Objects: Decision Frameworks for Project Management, Addison-Wesley, Reading MA, 1995

A comprehensive book on project management and object technology adoption, with a lot of case study information. Covers more ground than [Booch].

[Graham]

Graham I., Migrating to Object Technology, Addison-Wesley, Wokingham UK, 1995

One of the best sources for rules, including a shot at incorporating fuzzy logic to object systems. Has a detailed description of an object-oriented development process. Useful ideas on use-cases.

[Jacobson]

Jacobson I., Christerson M., Jonsson P. and Övergaard G. (1992) Object-Oriented Software Engineering: a use case driven approach, Addison-Wesley.

The book that started the charge of use-cases into object methods, as a result a very influential book. The treatment of use-cases is still the principal value of the book. Many of the modeling techniques have been assimilated into the UML

[Jacobson BPR]

Jacobson, I., Ericsson, M. and Jacobson, A. The Object Advantage: business process engineering with object technology, Addison-Wesley, 1995.

Jacobson applies use-case to business. There is an air of opportunism and stretching what is really a computer technique, but anyone who uses use-cases ought to think about the wider business use-cases.

[Martin]

Martin, R.C. Designing Object-Oriented C++ Applications Using the Booch Method, Prentice Hall, Englewood Cliffs, NJ, 1995.

Contains a series of large (for a book) and detailed case studies. For each study he takes you through design using class diagrams, interaction diagrams, and package diagrams using Booch's method. From the design he shows you how to get to C++. This book has the best treatment of package diagrams that I have seen.

[Meyer]

Meyer B. Object-Oriented Software Construction, Prentice Hall, Englewood Cliffs, NJ, 1997

Now updated, and huge, but despite it's size it is still a valuable book on object technology from a software engineering standpoint. Its most important feature is its discussion of design by contract.

[Odell foundations]

Martin, J. and Odell, J. Object oriented methods: a foundation, (UML Edition), Prentice Hall, Englewood Cliffs, NJ, 1995.

A very important book because it looks at objects from a conceptual perspective. As such it is one of the few books that really looks at objects from an analysis perspective. Includes a good description of multiple and dynamic classification, and details on event diagrams - the basis of UML's activity diagrams. It has recently been updated to the UML notation and is thus one of the first deep books to use

the UML.

[Odell pragmatics]

Martin, J. and Odell, J. Object oriented methods: pragmatic considerations, Prentice Hall, Englewood Cliffs, NJ, 1996.

Looks at how to get from models to code, reuse, and method engineering - a way of building custom methods for development organizations.

[PLoP 1 and 2]

Coplien, J.O. and Schmidt, D.C. Pattern Languages of Program Design, Addison-Wesley, Reading, MA, 1995.

Vlissides, J.M., Coplien, J.O. and Kerth, N.L. ed.Pattern Languages of Program Design 2, Addison-Wesley, 1996.

The procedings from the patterns conference. A wide range of patterns covering the whole field of software.

[Reenskaug]

Reenskaug, T., Wold, P. and Lehne, O.A. Working with objects: the OOram software engineering method, Manning Publications, Greenwich, CT, 1996.

Takes a different slant by looking at roles in interactions and then synthesizing the roles into classes. Reenskaug gives guidelines for performing safe synthesis: which ensures that the synthesized model will work correctly without analyzing the synthesized model. Usually this synthesis is done as part of modeling without any explicit separate step, so most methods concentrate on classes rather than roles.

[Rumbaugh]

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991) Object-Oriented Modeling and Design, Prentice Hall.

In this book Rumbaugh et al launched OMT, possibly the most popular of the OO methods, and a key foundation for the UML. As such this one of the most read descriptions of modeling with good descriptions of class diagrams and state diagrams. To be frank though, it is now too old and out of date.

[Shlaer and Mellor recursive design]

Shlaer S and Mellor SJ, Recursive Design of an Application Independent Architecture, IEEE Software, 14:1 (1997), p61-72

The best discussion so far of Shlaer/Mellor's recursive design approach and translation.

[Wirfs-Brock]

Wirfs-Brock R., Wilkerson B. and Wiener L. (1990) Designing Object-Oriented Software, Prentice Hall.

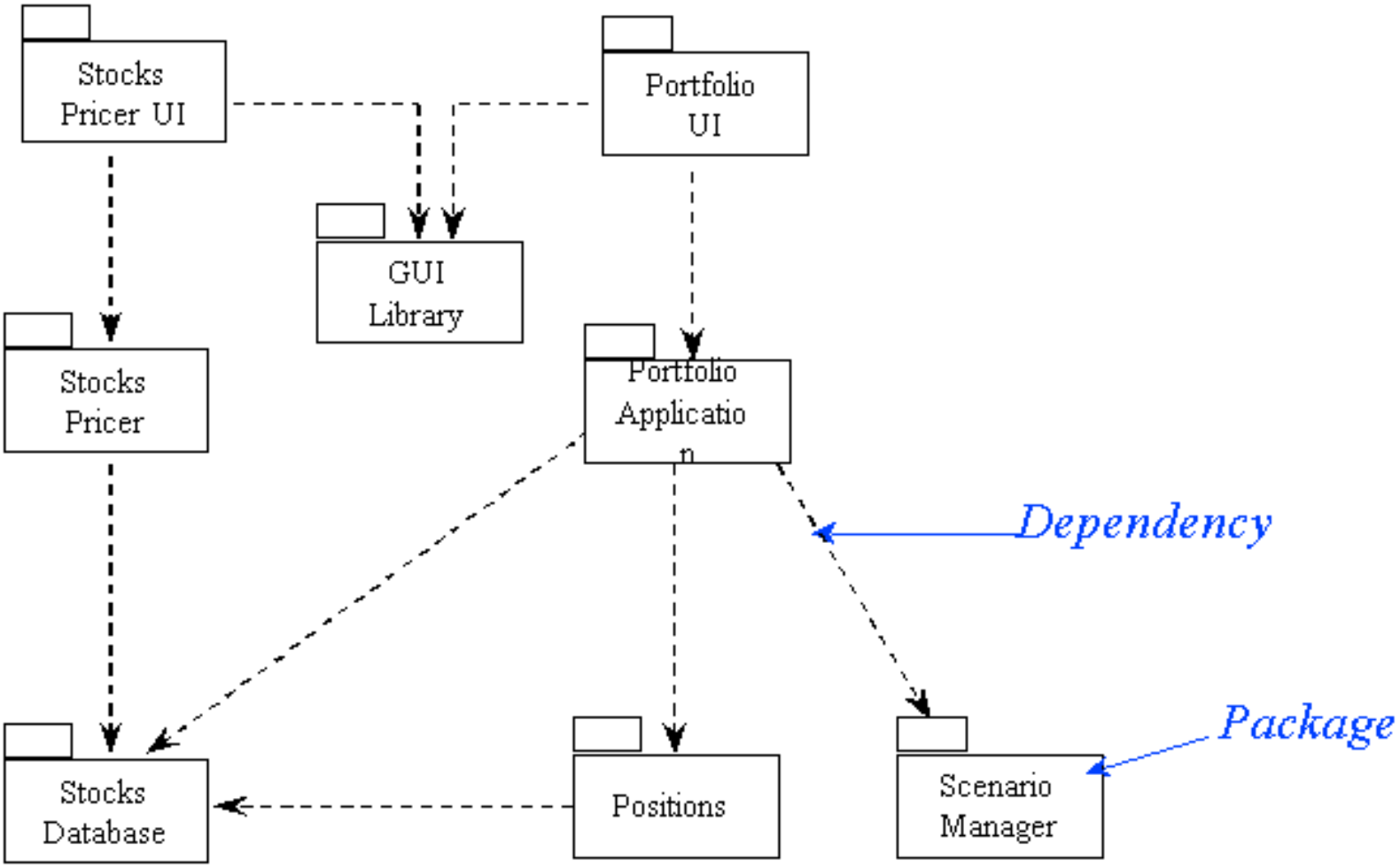
This book is really too old to be here, and yet it has aged really well. It has aged because of its emphasis on responsibilities and the basis of object design - even today a lesson that is often not fully understood. A good source of information about CRC cards.

|previous|contents|next|

Package Diagrams

Big systems offer special challenges. Draw a class model for a large system, and it is too big to comprehend. There are too many links between classes to understand. You could just chop the diagram into several pages, but while this makes looking at the diagram easier, the underlying software is just as likely to have lots of links that make the code difficult to understand and fragile.

A useful technique to handle this is that of UML's packages: which is based on Booch's class categories. In this technique each class is put into a single package. If a class wishes to use another class in the same package, all is well. If a class wants to use a class in a different package it must draw a dependency to that package. The overall picture of the system is the picture of packages and their dependencies, the aim is to keep the dependencies down to a minimum.



Within a package, classes can be public or private. Public classes are seen by those packages which have visibility, private types can only be used by classes within the same package. Packages can be made global, in which cases all other packages have visibility to them. This is necessary for general components such as integers, strings, and collections.

An important part of this approach is that dependencies are not transitive. By this I mean that if Portfolio UI has a dependency to Portfolio Application, and Portfolio Application has a dependency to Positions, it is not the case the Portfolio UI can use classes in Positions. Indeed the whole point of this system is a layered architecture where Portfolio Application is hiding the classes in Positions from Portfolio UI.

This lack of transitivity is important and is an important difference between package dependencies and both the

includes in C++ and the pre-requisites in Smalltalk's Envy. Compilation pre-requisites have to be transitive, but a good package architecture uses layering. Package dependencies are the same as Java's packages and import statement (which is not transitive). It is stronger than Java in that a dependency exists if a class is used without an import statement.

When to Use Them

Packages are a vital technique for any large scale system. You can manage quite well without them for small systems. They are less important for Smalltalk which is a dynamic environment with tools to quickly find method level dependencies.

The UML does not treat package diagrams as a separate technique, rather it treats them as icons on a class diagram. I prefer to treat them as a separate technique, but it is often useful to combine them by showing classes and packages on the same diagram.

Where to Find Out More

In all this technique is one of the least discussed in methods. [Booch] is the one who generally comes over as the most concerned, but his descriptions of his techniques are depressingly brief. The best description for this approach is that given by [Martin] who includes several examples of using packages (under the old Booch name of class categories) in the larger scale examples in his book. [Fowler] also discusses a few patterns of using this technique.

|previous|contents|next|

Design By Contract

Design By Contract is a design technique developed by Bertrand Meyer and a central feature of the Eiffel language that he developed. Design By Contract is not specific to Eiffel, however, and is a valuable technique for any programming language.

At the heart of design by contract is the assertion. An assertion is a Boolean statement. An assertion should never be false, and will only be false due to a bug. Typically assertions are only checked during debug and are not checked during production execution, indeed the program should never assume that assertions are being checked. Design By Contract uses three kinds of assertions: pre-conditions, post-conditions, and invariants.

Pre and post conditions are applied to operations. The post condition (it is easier to describe this first) is a statement of what the world should look like after an operation. For instance if we define the operation square on a number the post-condition would be of the form result = this * this (where result is the output and this is the object on which the operation was invoked). The post condition is a useful way of saying what we do, without saying how we do it, separating interface from implementation.

The pre-condition is a statement of how we expect the world to be before we execute the operation. Thus we might define a pre-condition for square of this >= 0. Such a pre-condition is saying that it is an error to invoke square on a negative number, and the consequences of doing so are undefined. On first glance this seems a bad idea, for we should put some check somewhere to ensure that square is invoked properly. The important question is who is responsible for doing so. The pre-condition makes it explicit that the caller is responsible for checking. Now without this explicit statement of responsibilities we can either get too little checking (because both assume the other is responsible) or too much (both check). Too much checking is a bad thing for it leads to lots of duplicate checking code which can significantly increase the complexity of a program. Being explicit about who is responsible helps to reduce this complexity. This danger that the caller forgets to check is reduced by the fact that assertions are usually checked during debugging and testing.

Out of this definition of pre and post condition we see a strong definition of an exception. An exception occurs when an operation is invoked with its pre condition satisfied, yet cannot return with its post-condition satisfied.

An invariant is a statement about a class. For instance a account class may have an invariant that says that balance == sum(entries.amount()). The invariant is 'always' true for all instances of the class. 'Always' means whenever the object is available to have an operation invoked upon it. In essence this means that the invariant is added to pre and post conditions of all public operations of the class. The invariant may become false during execution of a method, but should be restored by the time any other object can do anything to the receiver.

Assertions have a particular role in subclassing. One of the dangers of polymorphism is that you could redefine the subclass's operation in such a way as to be inconsistent with the superclass operation. The assertions stop you from doing this. The invariants and post-conditions of a class must apply to all subclasses. The subclasses can choose to strengthen these assertions but they cannot weaken them. The pre-condition on the other hand cannot be strengthened but may be weakened. This looks odd at first, but it is important to allow dynamic binding. You should always be able to treat a subclass object as if it

were an instance of the superclass. If a subclass strengthened its pre-condition then a superclass operation could fail when applied to subclass. Essentially assertions can only increase the responsibilities of the subclass. Pre-conditions are a statement of passing a responsibility on to the caller, you increase the responsibilities of a class by weakening a pre-condition. In practice all this allows much better control of subclassing, and helps you to ensure that subclasses behave properly.

Ideally assertions should be included in the code as part of the interface definition. Compilers should be able to turn assertion checking on for debugging and remove it for production use. Various stages of assertion checking can be used. Pre-conditions often give you the best chances of catching errors for the least amount of processing overhead.

When to Use Them

Design by Contract is a very valuable technique that you should use whenever you program. It is a great technique for helping you build clear interfaces. Only Eiffel supports assertions as part of its language, and Eiffel is (sadly) not a widely used language. It is possible to add mechanisms to C++ and Smalltalk to support some assertions. Java is rather more problematic.

Where to Find Out More

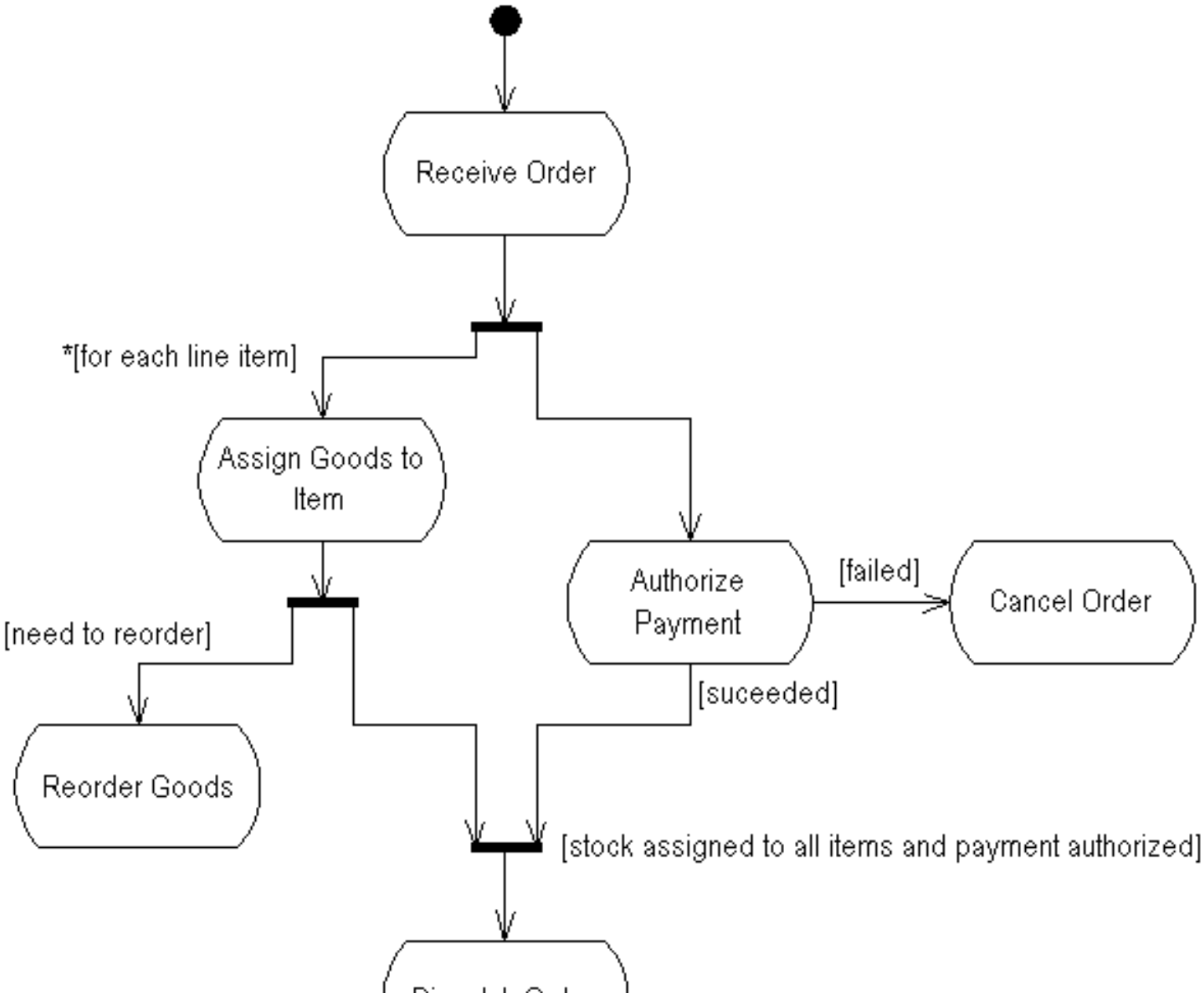
[Meyer] is a classic book on OO design that talks a lot about assertions. You can also get more information from ISE (Bertrand Meyer's company).

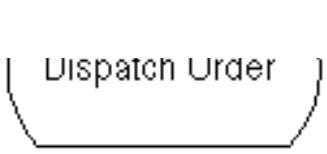
|previous|contents|next|

Activity Diagrams

For many people looking at the latest version of the UML, the most unfamiliar diagram is the new activity diagram. This is particularly unfamiliar since it is a diagram that was not present in works of either Booch, Jacobson, or Rumbaugh. In fact it is based upon the event diagram of Odell, although the notation is very different to that in Odell's books.

The activity diagram focuses on activities, chunks of process that may or may not correspond to methods or member functions, and the sequencing of these activities. In this sense it is like a flow chart. It differs, however, from a flow chart in that it explicitly supports parallel activities and their synchronization. In the diagram we see the receive order activity triggers both the authorize payment and the check line item activities. Indeed the check line item is triggered for each line item on the order. Thus if an order has three line items the receive order activity would trigger four activities in parallel, leading (at least conceptually) to four separate threads. These threads, together with other threads started by the receive supplies activity, are synchronized before the order is dispatched.





Note that this diagram does not have an end point. This is typical of activity diagrams that define a business process which synchronizes several external incoming events. You can think of the diagram of defining the reaction to that process, which continues until everything that needs to be done is done. You can also draw activity diagrams with a single start and end point; such procedural diagrams are more like the conventional procedure invocations. Such procedural diagrams can also use parallel behavior.

The biggest disadvantage of activity diagrams is that they do not make explicit which objects execute which activities, and the way that the messaging works between them. You can label each activity with the responsible object, but that does not make the interactions between the objects clear (that's when you need to use an interaction diagram). Often it is useful to draw an activity diagram early on in the modeling of a process, to help you understand the overall process. Then you can use interaction diagrams to help you allocate activities to classes.

When to Use Them

Activity diagrams are useful when you want to describe a behavior which is parallel, or when you want to show how behaviors in several use-cases interact.

Use interaction diagrams when you want to show how objects collaborate to implement an activity diagram. Use a state transition diagram to show how one object changes during its lifetime.

Where to Find Out More

The best source for information here is [Odell foundations]. He gives a thorough description of activity diagrams based on his original event diagram work.

|previous|contents|next|

CASE Tools

Structured methods have long fostered an industry of tools designed to help work with the methods. These tools, referred to as Computer Aided Software Engineering (CASE) tools have promised much, and delivered little. Sometimes it is said that such tools will one day replace programmers, perhaps they do not realize that this same claim was made for COBOL!

The key question is what is it that they do for you? Do they speed up the process of delivering software, do they help you build a higher quality design (that keeps you going quickly)? In my experience CASE tools capabilities boil down to two areas: design documentation and code generation.

Design Documents

Drawing graphical models on paper is time consuming and error prone, particularly as they need to be frequently revised. Thus dedicated drawing tools can support a similar level of support to software engineers that CAD tools provide in other disciplines. Unfortunately CASE tools often seem to operate on the assumption that the user is not proficient in modeling and thus constrain the user heavily. They allow you to draw only what they consider legal, which can get in the way of clear expression.

The best drawing tool for software engineering is a white-board (preferably a printing white-board). It is quick and easy to draw and erase, visible by a lot of people, and flexible. It encourages ideas to be tried out and discarded, and supports the group thinking so essential in analysis and design. No computer tool can match this facility. However for proper documentation some computer tool is very desirable. The first possibility to consider is a standard drawing tool such as Visio (you can get a good, free Visio stencil from Navision).. Such a thing is cheap and robust (by CASE tool standards) and runs on standard desktop platforms. The key element which many drawing tools lack is a facility to make lines stick to boxes, so that when the box moves the line moves too. The kind of limitations CASE tools inflict on drawings is frequently more of a disadvantage than it is advantage.

Beyond a drawing tool tools can offer a repository: essentially a database connected to the diagramming tool. Such a database should be able to list all classes in alphabetical order with a list of all of the characteristics of a class (associations, attributes, operations etc.) with the class. Typically this can consolidate models expressed on many diagrams.

The true closeness of the links between elements is an important factor here. A simple test is to change the name of a class on a model, and then see how the tool helps to keep everything up to date. Ideally a tool should change all references to the class wherever they appear in the repository. Often this does not occur. A particular area to watch is references to classes within text used for informal description. Usually these are not changed, and no tools are provided to help track the changes. Another area to consider is traceability links between different techniques. If an operation call on an interaction diagram refers to a class in its signature, will this change too? For this not to break down the tool must provide good support for a modeler to see the results of that "one little change" she just made.

Another important element to consider is how thus information will be accessed. Unless it is customary for everyone to work entirely on-line the repository and diagrams will need to be printed in a design document. Tools should allow the production of DTP quality design documents with the minimum of

effort. Again this is a common failing, many tools can take many days of effort to get the contents of the design into a presentable paper form.

The baseline here is the humble word-processor. This provides excelled design document facilities. The disadvantage of a word-processor is keeping it in step with changes in the diagrams, requirement time-consuming proof-reading. However this time can be less than the time taken to generate a design document from the CASE tool, and word-processors have very capable search and replace facilities which will capture all references to names. Another alternative can be to use a simple database.

Code-generation and execution

Code generation is probably the most hyped feature of CASE tools. The word code-generator is really another term for compiler and code-generators should be judged according to the same standards and with the same considerations (hence my cynical definition of a code-generator: a compiler that does not work properly).

One of the first questions that need to asked is: how much code is generated?. Most code generators are really only interface-generators: class definitions with attributes and operations. There is no code to actually carry out any processing. With these tools it is necessary to look carefully at how much of the interface is present. Are access and update operations generated for all the attributes? Are the operations merely names, or do they contain correct signatures?

The obvious limitation of this approach is that only interfaces are provided, and the bulk of the programming, the body code, still needs to be done. A second consideration is that of how evolution of the model and code is handled. A code-generator is of limited use if it cannot support the implementation consequences of changes to the analysis and design models. Many CASE tools seem to assume that code is only generated at the end of design. In real projects analysis, design and implementation evolve together. Effective interface generators need to provide close links between the CASE tool and the programming environment. Still these tools can be valuable providing they are used closely with the compiler.

Some tools support reverse engineering. This allows final code, preferably including code not generated by the CASE tool, to be read back into the CASE tool. In this way the tool provides a greater degree of comprehensibility than the source code. Naturally such an approach demands a very implementation oriented technique for it is unreasonable to expect a tool to reverse-engineer analysis.

Generating code is all very well, but what happens when you debug? Are you left in the programming environment with only mangled names to help you. Or can you actually debug the state diagrams to see what is going wrong? If you are going to use a tool for its code generation make sure you try out the whole process. You can't just generate and forget when you are using incremental development.

Tools vary in how much control you get over the code-generation process. In some tools they just generate what they think is a good idea. Does this correspond to your opinion? Some tools allow you to indicate how you want the code generated, either by providing hints or even by code snippets that indicate exactly how the code should be generated. Check how much customization the tool allows, and whether it is suitable for you.

|previous|contents|next|

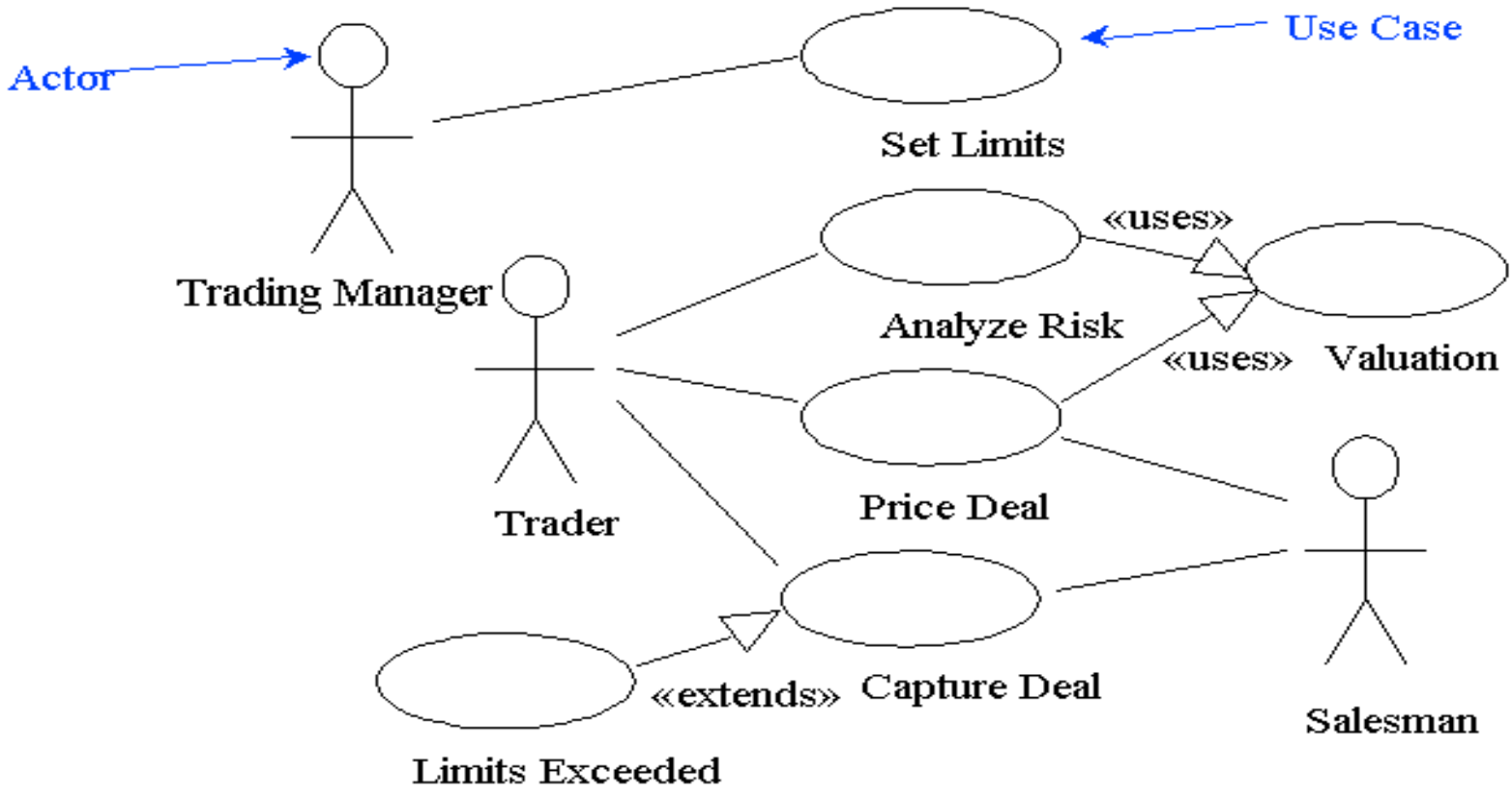
Use Cases

For a long time object developers struggled with where they got their information for their modeling and development from. They often assumed some requirements document, but were unsure of what such a document should look like. Here the impact of Ivar Jacobson's work on Use-Cases has been immense. The idea behind use-cases is not startling, and many people have used the general approach for a long time. Jacobson pushed this idea to the center of the stage, and that in itself is very important.

So what is a use-case? Now there's a question, use cases are one of those things that are difficult to define. [Jacobson] says they are when a user performs "a behaviorally related sequence of transactions in a dialogue with the system". Perhaps some examples will help. Use-cases for a word processor might include building an index or inserting a picture. In this way they correspond to menu commands. They can be quite large (building an index) or quite small (making some text bold). Often they might not involve a single command. A use case might be to ensure that the text in a document is consistently formatted; there is no command for this, but this is the use case that drives the need for style sheets.

This latter example introduces many of the difficulties of getting good use cases. The art is to identify the users' goals, not the system functions. One way of doing this is to treat a user's business task as a use case, and to ask how the computer system can support it.

Use-case diagrams provide a way of describing the external view of the system and its interactions with the outside world. In this way it resembles the context diagram of traditional approaches. In this representation the outside world is represented as actors. Actors are roles played by various people, or other computer systems. The emphasis on roles is important: one person may play many roles, and a role may have many people playing it. Use-cases are then typical interactions that the actor has with the system.



Use-cases can be seen as a large unstructured set or they can be structured in some way. [Jacobson] provides two structuring mechanisms. The first allows behavior used by several use-cases to be pulled out into a separate use-case which is used by the other ones. This is somewhat like pulling a common subroutine out which is shared by other routines. The second construct allows one use-case to extend another. The new use-case defines certain points in the original use-case at which it takes over with new behavior (it has been likened to a programming patch). Extensions are often used to show exception behavior and

special cases which would otherwise bloat the amount of use-cases in the model ([Graham] uses side-scripts for the same purpose). Use-cases act as the structuring mechanism for interaction diagrams. Typically an interaction diagram is drawn for each use-case in later design.

One of the big dangers of use-cases is that of structuring the software to mimic the use cases. Use-cases provide an external view of the system, the software is often structured in a completely different way. The biggest danger is that of turning each use-case into a procedural controller which acts upon simple data holders. When using use-cases remember that they are an external view only.

When to Use Them

Use cases a vital part of OO development. You should use them any time you want to understand the requirements of your system (that is all the time). Whether you use the diagrams is another matter. Use cases are valuable if just kept on a database as an unstructured list. Each use-case needs a name and a few paragraphs of description. They are central to planning the evolutionary development process. They should also drive system testing.

Where to Find Out More

I think the world is still waiting for a really good book on use-cases. Naturally [Jacobson] as a good source as the book that started it all, and I can't really leap to recommend a better choice. [Jacobson BPR] is useful for its accent on business process use-cases - which arguably should be used all the time. [Graham] also includes some good advice (he uses the term 'script'). One of the best papers I've come across on using well is by Alistair Cockburn and on his home page.

|previous|contents|next|

State Transition Diagrams

State transition diagrams have been used right from the beginning in object-oriented modeling. The basic idea is to define a machine that has a number of states (hence the term finite state machine). The machine receives events from the outside world, and each event can cause the machine to transition from one state to another. For an example, take a look at figure 1. Here the machine is a bottle in a bottling plant. It begins in the empty state. In that state it can receive squirt events. If the squirt event causes the bottle to become full, then it transitions to the full state, otherwise it stays in the empty state (indicated by the transition back to its own state). When in the full state the cap event will cause it to transition to the sealed state. The diagram indicates that a full bottle does not receive squirt events, and that an empty bottle does not receive cap events. Thus you can get a good sense of what events should occur, and what effect they can have on the object.

State transition diagrams were around long before object modeling. They give an explicit, even a formal definition of behavior. A big disadvantage for them is that they mean that you have to define all the possible states of a system. Whilst this is all right for small systems, it soon breaks down in larger systems as there is an exponential growth in the number of states. This state explosion problem leads to state transition diagrams becoming far too complex for much practical use. To combat this state explosion problem, object-oriented methods define separate state-transition diagrams for each class. This pretty much eliminates the explosion problem since each class is simple enough to have a comprehensible state transition diagram. (It does, however, raise a problem in that it is difficult to visualize the behavior of the whole system from a number of diagrams of individual classes - which leads people to interaction and activity modeling).

The most popular variety of state-transition diagram in object methods is the Harel Statechart as in Figure 1. This was introduced by Rumbaugh, taken up by Booch and adopted in the UML. It is one of the more powerful and flexible forms of state transition diagram. A particularly valuable feature of the approach is its ability to generalize states, which allows you to factor out common transitions (thus I can show that the break event applies to both full and empty states by creating the super-state of in-progress). It also has a flexible approach to handling processing. Processes that are instantaneous (i.e. cannot be interrupted) can be bound to the transitions or to the entry or exit of a state, these are called actions. Processes that are long (and can be interrupted) are bound to states, these are called activities. Transitions can also have a condition attached to them, which means that the transition only occurs if the condition is true. There is also a capability for concurrent state diagrams, allowing objects to have more than one diagram to describe their behavior.

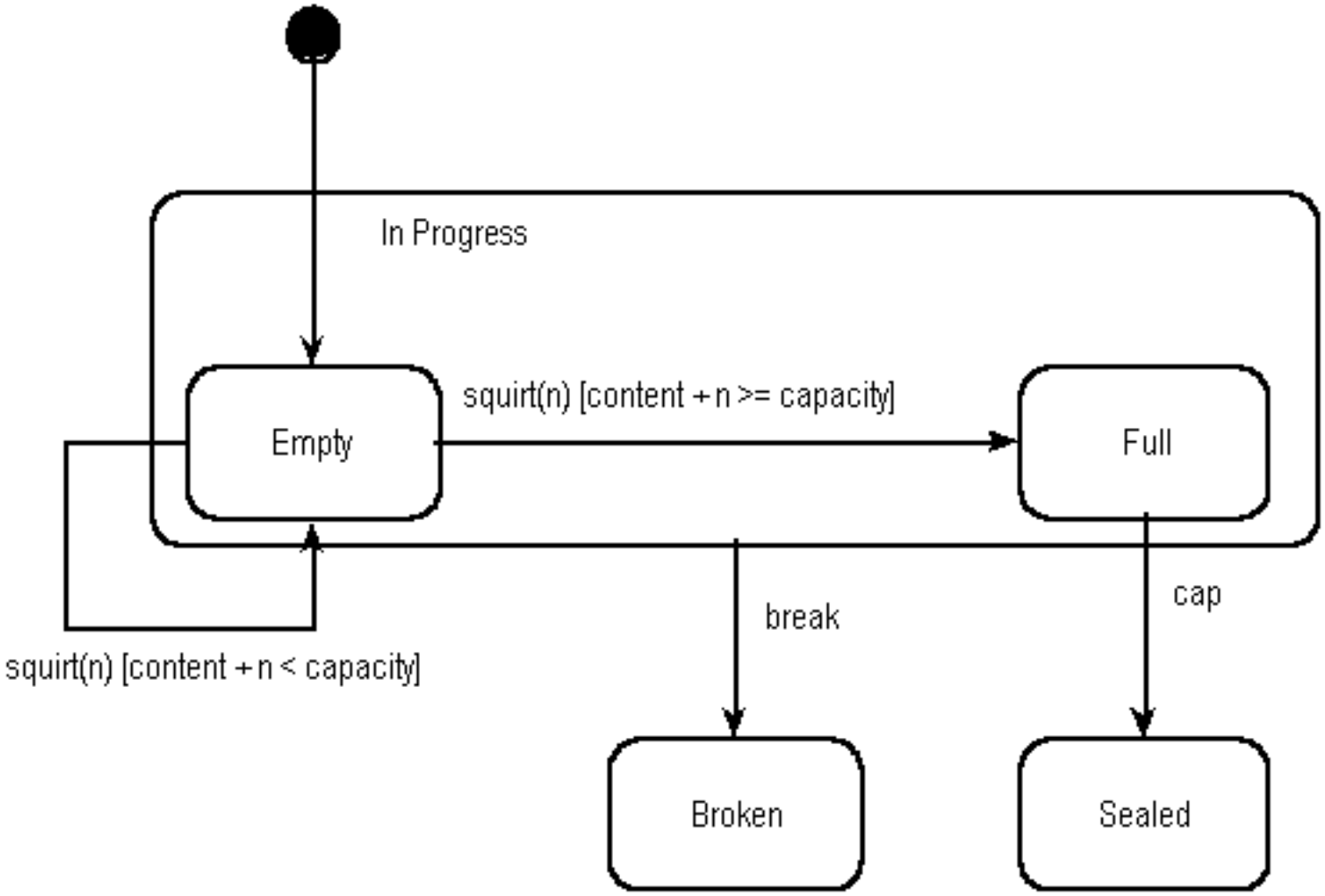


Figure 1: A Harel statechart

When to Use Them

State models are ideal for describing the behavior of a single object. They are also formal, so tools can be built which can execute them. Their biggest limitation is that they are not good at describing behavior that involved several objects, for these cases use an interaction diagram or an activity diagram.

People often do not find drawing state diagrams for several objects to be a natural way of describing a process. In these cases you can try either drawing a single state diagram for the process, or use an activity diagram. This defines the basic behavior, which you then need to split across a number of objects.

Where to Find Out More

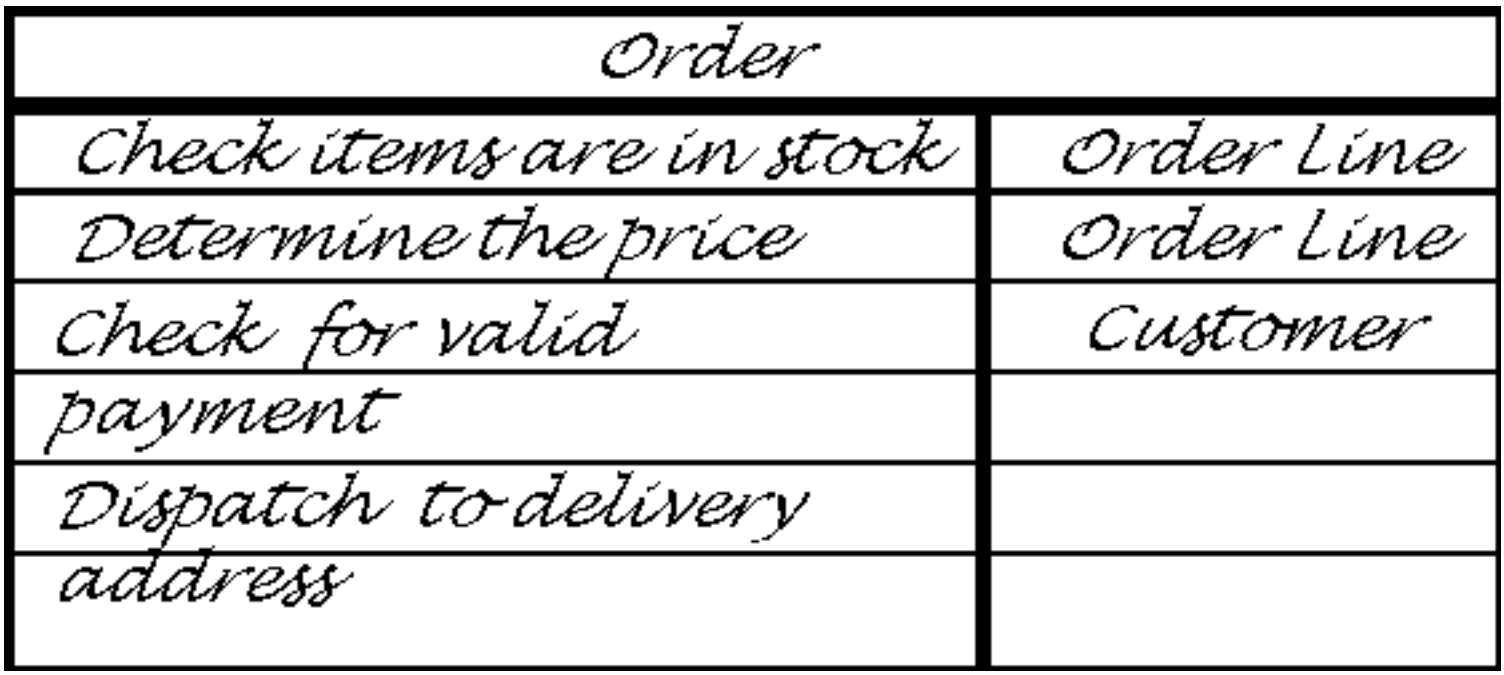
UML uses Harel statecharts, which have also become the most popular style of state diagrams in object methods. For an initial tutorial on them I would suggest either [Booch]. For a more in-depth treatment take a look at [Cook and Daniels]; they give a lot of good details on formalisms, an integration of design by contract, and a discussion of the use of state diagrams with subclassing.

|previous|contents|next|

CRC Cards

In the late 1980's one the biggest centers of object technology was the research labs of Tektronix in Portland Oregon. These labs were one of the main users of Smalltalk, and many key ideas in object technology were developed there. Two renowned Smalltalk programmers there were Ward Cunningham and Kent Beck. They were, and are, very concerned about how to teach people that same deep knowledge of how to use Smalltalk that they had gained. From this question of how to teach objects came the simple technique of CRC (Class-Responsibility-Collaboration) cards.

Rather than using diagrams to develop models, as most methodologists were doing, they represented classes on cards, using 4 X 6 index cards. Rather than indicating attributes and methods on the cards they write *responsibilities*. So what is a responsibility? It is really a high level description of the purpose of the class, the idea is to try to get away from a description of bits of data and process but instead to capture the purpose of the class in a few sentences. The choice of a card is deliberate - you are not allowed to write more than what will fit on the card.



The last letter refers to collaborators. With each responsibility you indicate which other classes you need to work with to fulfill it. This gives you some idea of the links between classes, but still at a high level.

CRC cards are an aggressively informal technique, they also encourage discussion. In a CRC session the developers will crowd around a table each person picking up a card as they describe how the class participates in some use case. By giving the developers something to touch that represents the class, you make it easier talk through the designs. By accenting responsibilities instead of data and methods you help them get away from dumb data holders and ease them towards understanding the higher level behavior of the class.

When to Use Them

Some people find CRC cards to be wonderful, others find CRC leave them cold. I certainly think you should try them out, to see if the team likes working with them. Use them particularly if you find teams are getting bogged down in two many details too early, or seem be identifying classes which seem cluttered and don't have clear definitions.

Use class diagrams and interaction diagrams to capture and formalize the results of CRC modeling. If you are using class diagrams ensure each class has a statement of its responsibilities.

Beware of long lists of responsibilities: if it doesn't fit on a 4X6 card you are missing the point.

Where to Find Out More

The best book that describes this technique, and indeed the whole notion of using responsibilities, is [Wirfs-Brock]. It is an old book, but it has aged well. You can get hold of Beck and Cunningham's original paper online.

|previous|contents|next|

Interaction Diagrams

Interaction diagrams are models that describe how a group of objects collaborate in some behavior - typically a single use-case. The diagrams show a number of example objects and the messages that are passed between these objects within the use-case.

I'll illustrate the approach with the following simple use-case. In this behavior the order entry window sends a prepare message to an order. The order then sends prepare to each order line on the order. The order line first checks the stock item, and if the check returns true it removes stock from the stock item. If stock item falls below the reorder level it requests a new delivery.

Interaction diagrams come in two forms, both present in the UML. The first form is the sequence diagram. In this form objects are shown as vertical lines with the messages as horizontal lines between them. This form was first popularized by Jacobson. The diagram below shows this form in its UML notation. The sequence of messages is indicated by reading down the page.

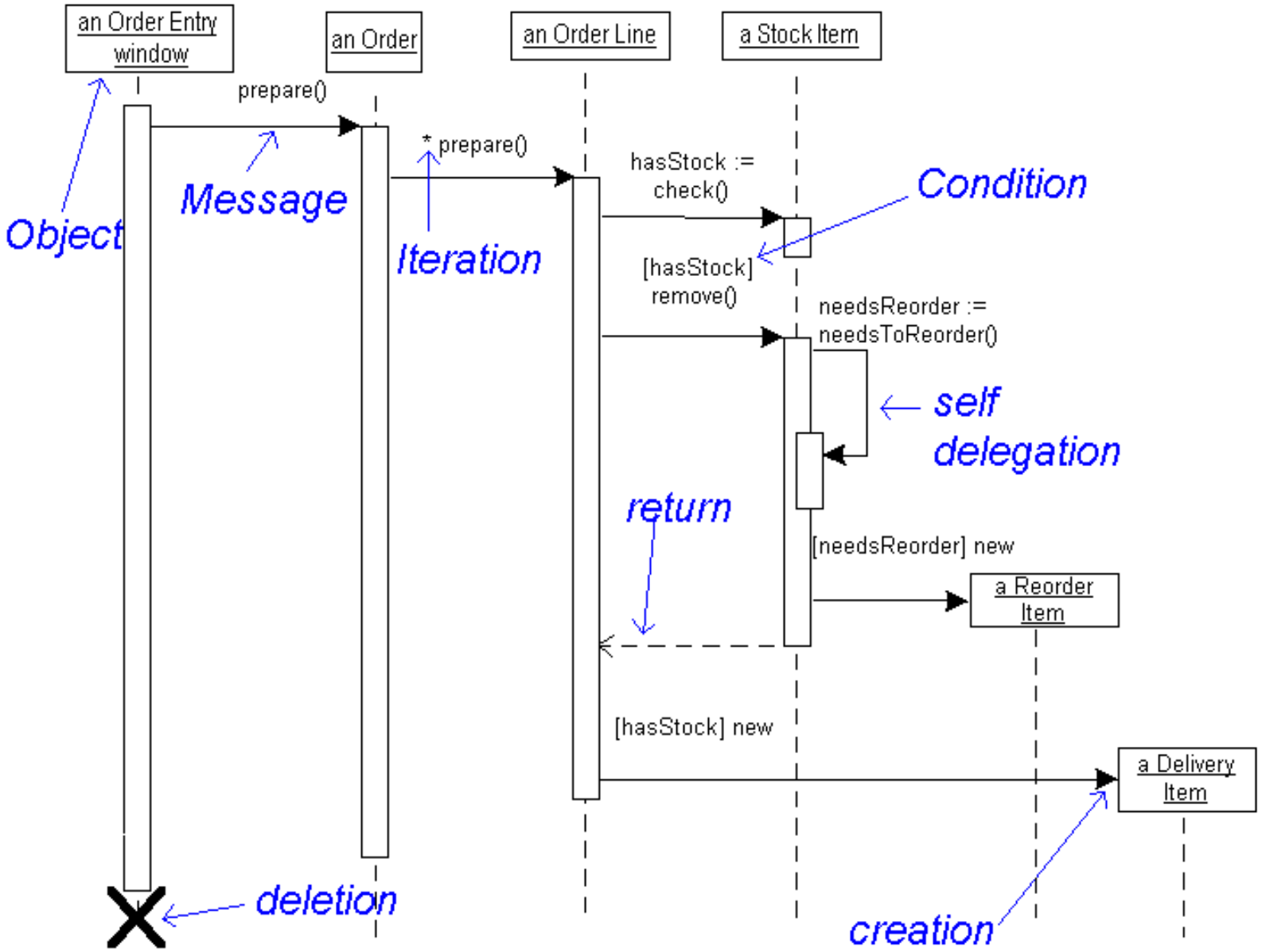


Figure 1: A sequence diagram.

The second form of the interaction diagram is the collaboration diagram. Here the example objects are shown as icons. Again arrows indicate the messages sent in the use case. This time the sequence is indicated by a numbering scheme. Simple collaboration diagrams simply number the messages in sequence. More complex schemes use a decimal numbering approach to indicate if messages are sent as part of the implementation of another message. In addition a letter can be used to show concurrent threads.

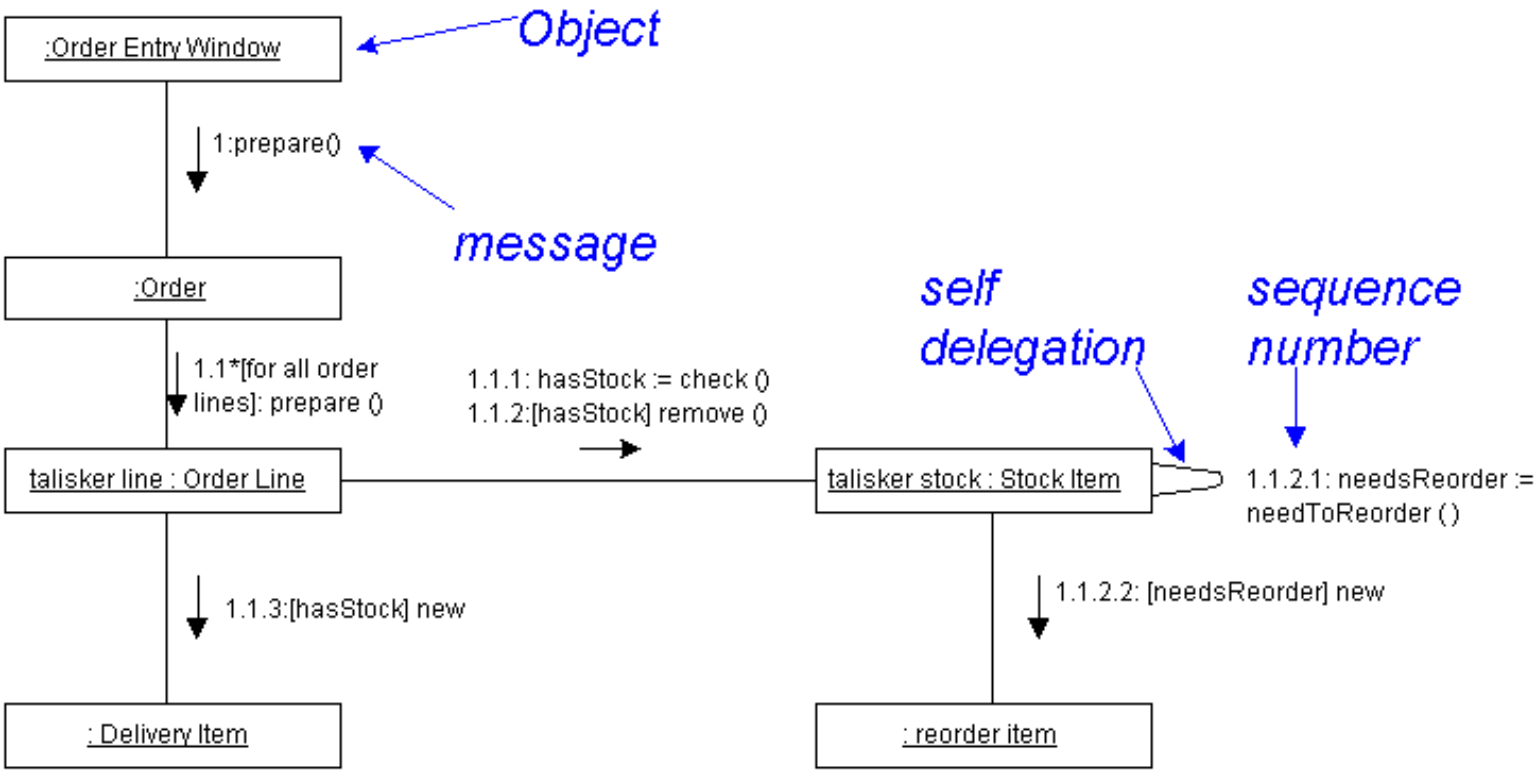


Figure 2: A collaboration diagram.

One of the great strengths of an interaction diagram is its simplicity. It is difficult to write much about interaction diagrams because they are so simple. They do, however, have weaknesses, the principal one is that although they are good at describing behavior: they do not define it. They typically do not show all the iteration and control that is needed to give an computationally complete description. Various things have been done to try and remedy this in the UML.

I have mixed feelings about these trends towards greater executability. To me, the beauty of interaction diagrams is their simplicity, and much of these additional notations lose this in their drive to computational completeness. Thus I would encourage you not to *rush* to the more complex forms of interaction diagrams, you may find that the simpler ones give you the best value.

When to Use Them

Interaction diagrams should be used when you want to look at the behavior of several objects within a single use case. They are good at showing the collaborations between the objects, they are not so good at precise definition of the behavior.

If you want to look at the behavior of a single object across many use-cases, use a state transition diagram. If you want to look at behavior across many use cases or many threads, consider an activity diagram.

Where to Find Out More

Most books on object-oriented modeling discuss some form of interaction diagram. Notations are currently very varied, but they should settle down to the UML in due course. A safe suggestion is that of [Booch] which gives you a simple outline, which is really all you need.

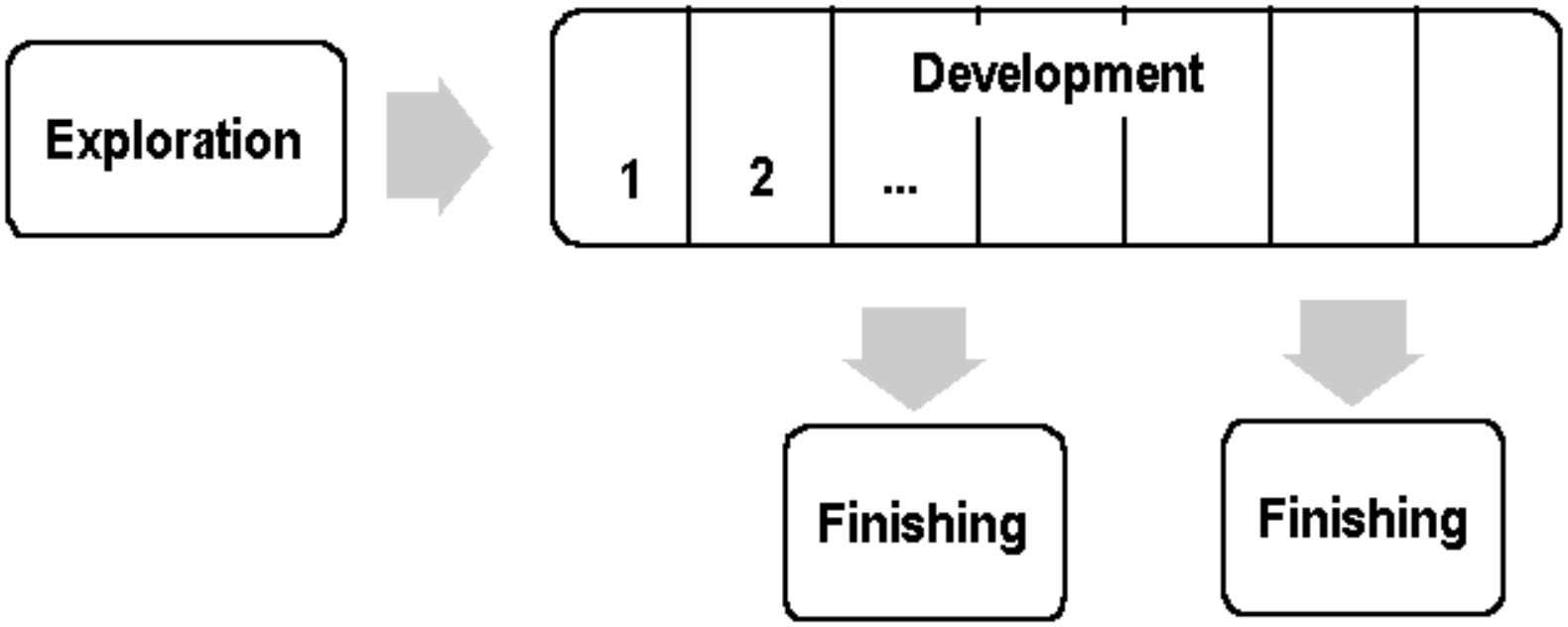
|previous|contents|next|

Incremental Development

Early books on object-oriented development did not say much about the process of developing an object-oriented system. Developers used to rigid methodological steps found little to guide, let alone constrain them. Over the last few years authors have paid much more attention to process. Several authors produce a detailed list of suggested activities with deliverables. In detail their processes look very different, however there is a common thread, and the commonality is often more significant than the differences.

All stress the fact that the waterfall life-cycle, with fixed phases of analysis, design, and construction, should join the Dodo. Instead they prefer a more incremental style. Often incremental (also called iterative or evolutionary) development is treated as an excuse for uncontrolled development. Actually incremental development can give much greater control than waterfall development, so although it takes a little getting used to, both managers and developers can get used to it.

The essence of evolutionary development is the development stage which is done as a series of increments. Each increment builds a subset of the full software system. An increment is a full lifecycle of analysis, design, coding, testing, and integration. It may be delivered to the user, but not necessarily. If it is delivered there may be some finishing: optimization, packaging and the like, after the build. The development stage is preceded by an exploration phase.



In the exploratory phase you are aiming to understand those areas that are most unknown to you. Typically the phase includes

- Finding use cases
- Developing an outline model of the problem domain
- Developing an outline architecture which describes how the major pieces of technology fit together

Usually prototyping will be done to accompany this work so that you can get a firm understanding of what the pieces entail.

This exploratory phase usually takes about 1/5 of the total length of the project. It is important that in this stage we are looking for outline models only. Detailed analysis and design is done as part the later incremental steps.

A good measure of when the exploration is complete is when the developers are able to estimate the time it will take to design and build each use case to nearest person/week.

The second stage is the incremental delivery of the software. The stage is broken into a series of increments. Each increment has the same duration and is marked by the release of a fully integrated product, with a full suite of tests, that satisfies some subset of the overall requirements. The division of requirements can be done by allocating use cases to the increments, so that each increment will implement a few of the overall amount of use cases. In this way the project has clearly visible and testable milestones that can be tracked by project's sponsors. It is this testability that makes a well planned incremental process more controlled than a waterfall project; there is no way to fool yourself with an untestable milestone such as 'analysis completed'.

How long should the increments be? Most writers aim for around 2-3 months, but there are some (including myself) that like to push for shorter increments. Step lengths of 2-3 weeks are very plausible, especially with rapid development languages such as Smalltalk. Each increment includes the analysis, design, coding, and testing of each of the use-cases assigned to that increment. The length of the increment should be fixed, if things are taking longer than they should then some use cases should be deferred to later, the date should never be altered. This gets a project into the habit of hitting a regular series of deadlines.

The increments should be structured to attack the largest risks first. Also the highest priority use cases need to be done first. A usable release to the customer should be done as soon as possible.

When to Use It

You should only use incremental development on projects you want to succeed.

Where to Find Out More

My preferred book on project management and process is [Cockburn]. It is short, sharp, to the point and covers all the really key things that need to be said.

|previous|contents|next|

Why Use Analysis and Design Techniques?

When it comes down to it, the real point of software development is cutting code. Diagrams are, after all just pretty pictures. No user is going to thank you for pretty pictures, what they want is software that executes. So when you are considering using these techniques it is important to ask yourself why you are doing it, and how it will help you when it comes down to the code. There's no proper empirical evidence to prove that these techniques are good or bad, but here I discuss the reasons that I often come across for using them.

Learning OO

A lot of people talk about the learning curve associated with OO, the infamous paradigm shift. In some ways the switch to OO is easy, in other ways there are a number of obstacles to working with objects, particularly in using them to their best advantage. Its not that its difficult to learn how to program in an OO language, it is that it takes a while to learn to exploit the advantages that object languages give you.

The techniques on this site were to some degree designed to help people do good OO, but different techniques have different advantages. One of the most valuable techniques for learning OO is CRC cards. They were designed for that primary purpose, and designed to be deliberately different to traditional design techniques. Their accent on responsibilities and their lack of complex notation makes them particularly valuable. Interaction diagrams are very useful because they make the message structure very explicit, and as such can highlight over-centralized designs, where one object is doing all the work.

Class diagrams are both good and bad for learning objects. They are comfortably similar to data models, and many of the principles that make a good data model make a good class model. The great problem is that it is easy to develop a class model that is very data oriented instead of responsibility oriented.

Patterns are an extremely important technique for learning OO, for they get you concentrate on good OO designs and to learn by following an example. Once you have got the hang of some simple modeling techniques, such as simple class diagrams and interaction diagrams, it is time to start looking at patterns. Another important process technique is that evolutionary development. Its not that this helps you learn in any direct way, but it is the key to exploiting OO effectively. If you do evolutionary development from the start, then you will learn in the context the right kind of process, and begin to see why designers suggest doing things the way they do.

Early on in using a technique you tend to do it by the book. Once you get comfortable you find it doesn't do certain things they way you would like. Don't hesitate to change things. The technique is there to serve you, not the other way round. If you find certain constructs unhelpful, then don't use them. If you need some construct that would be useful, then add. Just make sure you really understand what the construct means, both conceptually and when it comes to implementation. I like to look at any construct from the three perspectives: conceptual, specification, and implementation.

Communicating with Domain Experts

One of our biggest challenges in development is that of building the right system. This is made more difficult because we, with our jargon, have to communicate with users, who have their own more arcane jargon. (I did a lot of work in healthcare, and there the jargon isn't even in English!) Getting a good communication with good understanding of the user's world is the key to developing good software.

The obvious technique here is that of use cases. Use cases are the external picture of your system, and they go a long way to explaining what the system will do. A good collection of use cases is central to understanding what your users want. They also present a good vehicle for project planning as they control evolutionary delivery, which is itself a valuable technique since it gives regular feedback to the users of where the software is going.

As well as communicating about surface things, it is central to look at the deeper things. How do your domain experts understand their world. Class diagrams can be extremely valuable here, but use them in a *conceptual* manner. Treat each class as concept in the users mind, part of their language. The class diagrams you draw are then not diagrams of data, or of classes, but diagrams of the language of your users. [Odell foundations] is a good source for this kind of thinking with class diagrams. If processes are an important part of the users' world then I've found activity diagrams to be very useful. Since they support parallel processes they can help you get away from unnecessary sequences. The way they de-emphasize the links to classes, which can be a problem in later design, becomes an advantage in this more conceptual stage.

Understanding the Big Picture

As a consultant I often have to breeze into a complex project and look intelligent in a very short period of time. I find these kind of design techniques invaluable for that, for they give me an overall view of the system. A look at a class diagram can quickly tell me what kinds of abstractions are present in the system, and where the questionable parts are that need further work. As I probe deeper I want to see how classes collaborate ask to see interaction diagrams of key behaviors in the system.

If this is useful to me as an outsider, it is just as useful to the regular project team. Its easy to lose sight of the wood for the trees on a large project. With a few choice diagrams to hand, you can find your way around the software much more easily. To build a road-map use package diagrams at the higher levels to scope out a class diagram. When you draw a class diagram for a road-map take a specification perspective. Its very important to hide implementations with this kind of work. Don't document every interaction, just the key ones. Use patterns to describe the key ideas in the system, they help you to explain *why* your design is the way it is. It is also useful to describe designs you have rejected, and why you did that. I always end up forgetting those kind of decisions.

Translation

Translation is a term coined by Steve Mellor, a term he uses to emphasize the key difference between his approach and the majority of OO methods, which he refers to as elaboration methods. This difference covers the process by which you take a conceptual analysis model of a domain, and turn it into code in your particular implementation environment.

With elaboration you take a high level, analysis model of your domain and gradually add the details you need to turn it into working code. With translation you only detail the analysis model to the point that you have a rigorous and executable analysis model, but you do not add any implementation dependent detail. The design process instead involves deciding how to implement the *constructs* of the analysis model. These decisions are captured in a set of archetypes, essentially a specification for a compiler that will compile your analysis model into your implementation environment.

Translation's advocates claim the following advantages

- Separating the domain analysis from the implementation allows you to change implementation easily. You can work with a set of archetypes for one implementation environment while you are iterating the domain model, and switch to a different set for production. Tools are available that can interpret the translation style models, these are ideal for this process.
- Translation allows you to have more specialized skills. Those skilled in manipulating domain models concentrate in producing a good analysis model without worrying about the implementation issues. Different people can specialize on developing archetypes independently of any specific domain.
- Once you have developed a good set of archetypes you can reuse them on future projects that have the same implementation environment. A popular implementation environment would have archetypes available that you could buy for it.

In theory translation has a lot going for it. The crunch question is how much the theory works out in practice. Translation will only realize its benefits if the detailed analysis models are easier to manipulate than conventional programming languages, and if enough good archetypes are available. Translation's detractors claim that neither is true. To do detailed enough analysis models requires detailed techniques, usually relying on state machines, that are in themselves complicated programming. Tools may generate target code, but debuggers are just as important and much harder to produce. Translation methods end up as proprietary languages, and increasingly companies are looking to more standard languages that do not lock themselves into a single vendor.

As soon as we enter this subject we enter a commercial battleground. Steve Mellor's company Project Technology produce tools and consulting to support a translation based approach. They claim great success with this. Naturally their competitors question these claims.

Project Technology are not the only ones who use translation, however. ObjectTime and the ROOM method. Their modeling language is more complex and their supporters claim this better supports this style of working. They are also more focused on real-time developments. Project Technology's principal experience is with real-time, but they claim their approach is generally applicable.

When to Use It

There is no strong evidence either way on whether translation lives up to its claims. Certainly it is worth taking a look at. Evaluate specific tools against your project's needs. Are suitable archetypes available for your environment? Do you think it is worth building such archetypes? Take a careful look at the debugging process, debuggers are usually harder to build than compilers, yet are essential to serious development.

Where to Find Out More

The most vocal proponent of this approach is Steve Mellor, and he and Sally Shlaer wrote an IEEE Software that describes their approach [Shlaer and Mellor recursive design], apparently there is also a book in the works. Naturally Project Technology is good source of information on this topic. So are ObjectTime for their brand of translation. [Odell pragmatics] discusses some general approaches to building archetypes (he calls them 'Design Templates').

|previous|contents|next|

Patterns

Most of the analysis and design techniques that I talk about here are about giving you guidance on how to express an object-oriented design. Patterns look instead at the results of the process - example models. Many people have commented that projects have problems because the people on them were not aware of designs that are well-known to those who have more experience. Patterns describe common ways of doing things. They are collected by people who spot some repeated themes in designs they come across. They then take each of these themes and describe them so that other people can read the pattern and see how to apply it.

Its easier to use an example. Say you have some objects running in a process on your desktop and they need to communicate with some other object that is running in another process, maybe o

Refactoring

Have you come across the principle of Software Entropy? It suggests that programs start in a well designed state, then as new bits of functionality are tacked on they gradually lose that well designed structure, eventually deforming into a mass of spaghetti. Part of this is scale, you write a small program that does a specific job well, people ask you to enhance the program and it gets more complex. Even if you try to keep track of this design this can still happen, as you do more the program gets more complex.

One of the reasons this happens is because when you are adding a new feature to a program you are building on top of the existing program, often in a way that the existing program was not intended for. In such a situation you can either redesign the existing program to better support your changes, or you can work around them in your additions. Although in theory it is better to redesign your program, this usually results in extra work because any rewriting of your existing program will introduce new bugs and problems. Remember the old engineering adage: 'if it aint broke, don't fix it'. However if you don't redesign your program the additions will be more complex than they should be. Gradually this extra complexity will exact a stiff penalty. Thus you have a trade-off: redesigning causes short-term pain for a longer term gain. Schedule pressure being what it is, most people prefer to put their pain off to the future.

Refactoring is a term used to describe techniques that reduce the short-term pain of redesigning. When you are refactoring you are not changing the functionality of your program, you are changing its internal structure in order to make it easier to understand and work with. Refactoring changes are usually small steps, renaming a method, moving a field from one class to another, or consolidating two similar methods in a superclass. Each step is tiny, yet in a couple of hours of these small steps you can do a power of good to a program.

Refactoring is made easier by the following principles

- Do not do both refactoring and adding functionality at the same time. Put a clear separation between the two when you are working. You might swap between them in short steps: half an hour refactoring, an hour adding new function, half an hour refactoring the code you just added.
- Make sure you have good tests before you begin refactoring. Run the tests as often as possible. That way you will know quickly if your changes have broken anything.
- Take short deliberate steps: moving a field from one class to another, fusing two similar methods into a superclass. Refactoring often involves making many localized changes that result in a larger scale change. If you keep your steps small, and test after each step, you will avoid prolonged debugging.

When should you refactor?

- When you are adding function to your program and you find the old code getting in the way. When that starts becoming a problem, stop adding the new function and instead refactor the old code
- When you are looking at code and having difficulty understanding it. Refactoring is a good way of helping you understand the code and preserving that understanding for the future.

Often you will find yourself wanting to refactor code that someone else has written. When you do this, do it with the author of the code. It is difficult to write code in a way that others can easily understand it. The best way to refactor is to have someone else try to understand the code and combining your understanding with their unfamiliarity.

When to Use It

Refactoring is a much under-used technique. It has only just begun to be recognized, mainly in the Smalltalk community. However I believe it is a key technique in improving software development, whatever enviornment you use. Ensure your developers understand how to do refactoring in a disciplined way, get your mentor to teach them the techniques.

Where to Find Out More

Because refactoring is still very new as a technique, there is little that has been written on it. William Opdyke's PhD thesis is probably the largest treatment of the subject, but it is geared to automatic refactoring tools, rather than techniques that humans can use now. Kent Beck is one of the foremost exponents of refactoring, his patterns book [Beck] includes many patterns that are central to refactoring. If you use VisualWorks Smalltalk you can download Refactory, a tool that supports refactoring.

|previous|contents|